Error Handling

Play+ Error Handling Helper : playerror Introduction ■ In the Play+ ecosystem, we believe that errors are opportunities to build trust, not failures to hide. This helper is based on the concept of Resilience by Design , where error handling transforms from a reactive, defensive practice into a proactive, trust-building experience. The playerror helper provides a comprehensive, user-centric approach to managing errors throughout your application. It automatically enriches errors with context, classifies them by severity, and provides clear recovery paths. This aligns directly with our Intuitive pillar by making error handling predictable and user-friendly, our Adaptive pillar by gracefully handling different error scenarios, and our Engaging pillar by maintaining user trust even when things go wrong. Package Info ■ The Play+ error handling helper and its associated configurations are included by default in the Golden Path starter kit. Package / Path Golden Path (Recommended) Pre-installed /system/play.error.ts Uplift Path npm install @playplus/error Folder Reference ■ The error handling helper and its configuration follow our standardized folder structure for core system logic. File / Directory Purpose & Guidelines system/play.error.ts The core error handling service. It provides utilities for error reporting, classification, and user feedback. config/play.error.config.json User-overridable configuration for error messages, PII redaction, and handling strategies. src/app/core/global-error-handler.ts Angular global error handler that integrates with the Play+ error system. src/app/services/playerror.service.ts Angular service wrapper that provides component-specific error handling utilities. src/app/components/error-boundary/ Error boundary components for React-style error isolation. Helper - Pillars Alignment ■ The playerror helper is a direct implementation of our core design pillars, focused on building user trust through intelligent error handling. Pillar How This Helper Aligns Intuitive Primary Pillar: Provides clear, actionable error messages and predictable recovery paths that users can easily understand and follow. Adaptive Gracefully handles different error scenarios (network, validation, server) with appropriate responses and fallback strategies. Engaging Maintains user engagement by preserving work when possible and providing helpful recovery options instead of frustrating dead ends. Helper Overview ■ The playerror toolchain is a comprehensive error management system that automates the complex aspects of error handling while providing developers with simple, powerful tools. Instead of manually writing try-catch blocks and error reporting logic, developers can rely on the Play+ system to handle errors intelligently. It automates error quality control in several key ways: Automatic Error Capture : Global handlers catch uncaught exceptions and route them through the Play+ system Contextual Intelligence : Automatically enriches errors with user, session, and application context Smart Classification : Determines error type and severity to route to appropriate handling strategies User-Centric Feedback : Transforms technical errors into clear, actionable user messages Recovery Paths : Provides clear next steps for every error scenario The goal is to make robust error handling the default path, not an afterthought. Config Options ■ Global error handling settings are managed in config/play.error.config.json . These values control error messages, PII redaction, and handling

strategies. Config Key Table ■ Config Key Default Value Description Recommended Value defaultMessages.server "We're fixing it!" Default message for server errors. Custom message for your app defaultMessages.notFound "Item not found." Default message for 404 errors. Custom message for your app defaultMessages.auth "Please log in again." Default message for authentication errors. Custom message for your app defaultMessages.forbidden "You lack permission." Default message for 403 errors. Custom message for your app defaultMessages.timeout "Connection timed out." Default message for timeout errors. Custom message for your app defaultMessages.network "Check your internet connection." Default message for network errors. Custom message for your app defaultMessages.rateLimit "Too many requests." Default message for rate limiting errors. Custom message for your app defaultMessages.validation "Please check your input and try again." Default message for validation errors. Custom message for your app defaultMessages.unknown "Something went wrong. Please try again." Default message for unknown errors. Custom message for your app piiFields ["password", "email", "token", "apiKey", "secret", "authorization", "cookie", "sessionId"] Fields to redact from error logs. Add any additional sensitive fields enableAutoReporting true Whether to automatically report errors to monitoring systems. true enableUserFeedback true Whether to show user-friendly error messages. true enableRecoveryOptions true Whether to provide recovery options for errors. true Helper Methods ■ Core Methods ■ Method Name Description Signature report Reports non-critical errors with context and user feedback. report(error: Error, context?: ErrorContext): void reportCritical Reports critical errors with immediate user notification and recovery options. reportCritical(error: Error, context?: ErrorContext): void showToast Shows a toast notification with error message. showToast(message: string, type: 'error' | 'warn' | 'info'): void showModal Shows a modal dialog with error details and recovery options. showModal(title: string, message: string, actions: ErrorAction[]): void showInline Shows inline validation error for form fields. showInline(elementId: string, message: string, type: 'error' | 'warn'): void withContext Creates a scoped error reporter with predefined context. withContext(context: ErrorContext): ErrorReporter Angular Integration ■ PlayErrorService ■ Angular service wrapper that integrates with Play+ logging and provides component-specific error handling utilities. import { PlayErrorService } from '../services/playerror.service' ; @ Component ({ ... }) export class MyComponent { constructor (private playError : PlayErrorService) { } async fetchUserData ( ) { try { const user = await this . apiService . get ('/api/users/123') ; return user ; } catch (error) { this . playError . handleApiError (error , { action : 'fetchUserData' , endpoint : '/api/users/123' } , ( ) => { // Retry action this . fetchUserData ( ) ; }) ; throw error ; } } } Global Error Handler ■ Automatically catches and processes uncaught exceptions throughout the application. // src/app/core/global-error-handler.ts import { ErrorHandler } from "@angular/core" ; import { playerror } from "../../../system/play.error" ; export class GlobalErrorHandler implements ErrorHandler { handleError (error : Error) : void { playerror . report (error , { component : "Global" , action : "uncaught" , }) ; } } Provides React-style error isolation for complex components. < app-error-boundary level = " component " errorTitle = " Component Error " errorMessage = " This component encountered an error. " [showRetry] = " true

" (onRetry) = " retryComponent() " > < app-complex-component > </ app-complex-component > </ app-error-boundary > React: Basic Error Reporting ■ This example shows how to report errors with context and provide user feedback. import React from "react" ; import { playerror } from "../system/play.error" ; function UserProfile ({ userId }) { const [user , setUser] = useState (null) ; const [loading , setLoading] = useState (true) ; useEffect (( ) => { async function fetchUser ( ) { try { const response = await fetch (` /api/users/ ${ userId } `) ; if (! response . ok) { throw new Error (` HTTP ${ response . status } : ${ response . statusText } `) ; } const userData = await response . json ( ) ; setUser (userData) ; } catch (error) { playerror . report (error , { component : "UserProfile" , action : "fetchUser" , userId : userId , }) ; playerror . showToast ("Unable to load user profile. Please try again." , "error") ; } finally { setLoading (false) ; } } fetchUser ( ) ; } , [userId]) ; if (loading) return < div > Loading... </ div > ; if (! user) return < div > User not found </ div > ; return (< div > < h1 > { user . name } </ h1 > < p > { user . email } </ p > </ div >) ; } Angular: API Error Handling ■ This example shows how to handle API errors with retry functionality and user feedback. // user.service.ts import { Injectable } from "@angular/core" ; import { PlayErrorService } from "../services/playerror.service" ; @ Injectable ({ providedIn : "root" , }) export class UserService { constructor (private http : HttpClient , private playError : PlayErrorService) { } async fetchUserData (userId : string) : Promise < User > { try { const user = await this . http . get < User > (` /api/users/ ${ userId } `) . toPromise ( ) ; return user ; } catch (error) { this . playError . handleApiError (error , { action : "fetchUserData" , endpoint : ` /api/users/ ${ userId } ` , userId : userId , } , ( ) => { // Retry action return this . fetchUserData (userId) ; }) ; throw error ; } } } import { playerror } from "../../../system/play.error" ; // Report non-critical errors playerror . report (new Error ("Something went wrong") , { component : "UserService" , action : "fetchUserProfile" , userId : "user-123" , }) ; // Report critical errors playerror . reportCritical (new Error ("Critical system failure") , { component : "AuthService" , action : "validateToken" , }) ; // Show toast notification playerror . showToast ("Please check your internet connection" , "warn") ; // Show modal for critical errors playerror . showModal ("Connection Error" , "Unable to connect to the server. Please try again." , [{ label : "Retry" , action : ( ) => retryConnection ( ) , primary : true , } , { label : "Go Offline" , action : ( ) => enableOfflineMode ( ) , } ,]) ; // Show inline validation errors playerror . showInline ("email-error" , "Please enter a valid email address" , "error") ; // Create scoped error reporter const userReporter = playerror . withContext ({ component : "UserService" , userId : "user-123" , }) ; // Use scoped reporter userReporter . report (new Error ("Failed to update profile") , { action : "updateProfile" , field : "email" , }) ; Additional Info ■ Why We Created This Helper ■ Error handling is critical for user experience, but implementing it correctly is complex and error-prone. Without a dedicated system, developers would need to: Manually write try-catch blocks for every async operation Create custom error reporting logic for each component Implement user feedback mechanisms for different error types Handle PII redaction and security concerns manually Build recovery paths and retry mechanisms from scratch This leads to inconsistent error handling and poor user experiences. The playerror helper automates these common patterns and provides a simple, consistent API that makes robust error handling the default path. Error Classification ■ The system automatically classifies errors into three

categories: System Errors (Critical) ■ Network failures : Connection timeouts, DNS failures Server errors : 5xx responses, service unavailable Runtime errors : JavaScript exceptions, memory issues Handling : Immediate reporting, user modal, retry options User Errors (Non-Critical) ■ Validation errors : Invalid input, missing required fields Business rule violations : Insufficient permissions, quota exceeded Authentication errors : Expired tokens, invalid credentials Handling : Toast notifications, inline feedback, guided correction Environmental Errors (Adaptive) ■ Offline state : No internet connection Slow connections : Timeout warnings Browser limitations : Storage blocked, feature unsupported Handling : Adaptive UI, cached data, transparent communication Best Practices ■ DO ■ Use contextual reporting : Always provide component and action context Classify errors appropriately : Use report() for non-critical, reportCritical() for critical Provide recovery paths : Every error should have a clear next step Test error scenarios : Simulate network failures, invalid inputs, server errors Monitor error trends : Track error rates, recovery rates, and user impact Use error boundaries : Wrap complex components to prevent cascading failures DON'T ■ Show technical errors to users : Never expose stack traces or technical details Throw errors without reporting : Always use playerror.report() or reportCritical() Ignore user context : Don't report errors without user and session information Use generic error messages : Provide specific, actionable feedback Forget recovery options : Every error should have a way forward Log sensitive data : PII is automatically redacted, but be mindful of additional context Security Considerations ■ PII Protection : Sensitive data is automatically redacted from logs Error Sanitization : Error messages are sanitized before display Secure Reporting : Error reports are sent over secure channels Access Control : Error logs are protected by appropriate access controls Forbidden Patterns ■ Console Error Logging ■ // DON'T: Use console.error for error reporting try { const result = await riskyOperation ( ) ; return result ; } catch (error) { console . error ("Operation failed:" , error) ; throw error ; } // DO: Use playerror.report() try { const result = await riskyOperation ( ) ; return result ; } catch (error) { playerror . report (error , { component : "RiskyComponent" , action : "riskyOperation" , }) ; throw error ; } Generic Error Messages ■ // DON'T: Show generic error messages playerror . showToast ("Something went wrong" , "error") ; // DO: Provide specific, actionable feedback playerror . showToast ("Unable to save your changes. Please check your connection and try again." , "error") ; Technical Error Exposure ■ // DON'T: Show technical errors to users playerror . showModal ("Error" , error . message , [ ]) ; // DO: Show user-friendly messages playerror . showModal ("Connection Error" , "Unable to connect to the server. Please try again." , [{ label : "Retry" , action : retryFunction , primary : true }]) ; Missing Recovery Options ■ // DON'T: Show errors without recovery paths playerror . showToast ("Operation failed" , "error") ; // DO: Provide clear next steps playerror . showModal ("Save Failed" , "Unable to save your changes." , [{ label : "Try Again" , action : retrySave , primary : true } , { label : "Save as Draft" , action : saveAsDraft } , { label : "Cancel" , action : cancel } ,]) ; Required Patterns ■ Contextual Error Reporting ■ // Always provide context when reporting errors playerror . report (error , { component : "UserService" , action : "updateProfile" , userId : "user-123" , field : "email" , }) ; Error Classification ■ // Use appropriate reporting method based on error severity if (isCriticalError (error)) { playerror . reportCritical (error , context) ; } else {

playerror . report (error , context) ; } User Feedback ■ // Always provide user-friendly feedback playerror . showToast ("Unable to load data. Please try again." , "error") ; Recovery Paths ■ // Provide clear recovery options playerror . showModal ("Connection Lost" , "Your connection was interrupted." , [{ label : "Reconnect" , action : reconnect , primary : true } , { label : "Continue Offline" , action : enableOfflineMode } ,]) ; Testing ■ Error Simulation ■ // Test network errors beforeEach (( ) => { spyOn (window , "fetch") . and . returnValue (Promise . reject (new Error ("Network error"))) ; }) ; // Test validation errors it ("should handle validation errors" , ( ) => { const errors = { email : { required : true } } ; component . playError . handleValidationError (errors , "test-form") ; expect (component . showInlineError) . toHaveBeenCalledWith ("test-form-email-error" , "This field is required." , "error") ; }) ; // Test error boundaries it ("should render fallback UI on error" , ( ) => { component . throwError ( ) ; expect (fixture . debugElement . query (By . css (".error-boundary"))) . toBeTruthy ( ) ; }) ; Error Monitoring ■ // Monitor error rates const errorRate = errorsPerMinute / totalRequestsPerMinute ; if (errorRate > 0.05) { // Alert on high error rate alertTeam ("High error rate detected") ; } // Monitor recovery rates const recoveryRate = successfulRetries / totalErrors ; if (recoveryRate < 0.8) { // Investigate low recovery rate investigateRecoveryIssues ( ) ; } Monitoring and Analytics ■ Error Metrics ■ Error Rate : Percentage of requests that result in errors Recovery Rate : Percentage of errors that users successfully recover from Mean Time to Recovery (MTTR) : Average time to resolve errors User Impact : Number of users affected by errors Error Distribution : Types and frequencies of different errors Alerting ■ // Critical error alerts if (error . critical && error . affectsMultipleUsers) { alertTeam ("Critical error affecting multiple users" , { error : error , affectedUsers : error . affectedUsers , priority : "P0" , }) ; } // Error rate alerts if (errorRate > threshold) { alertTeam ("Error rate exceeded threshold" , { currentRate : errorRate , threshold : threshold , priority : "P1" , }) ; } Integration with Other Play+ Systems ■ Logging Integration ■ // Errors are automatically logged with rich context playerror . report (error , context) ; // → Automatically logged via playlog with error level Security Integration ■ // Error context is automatically sanitized playerror . report (error , { userInput : maliciousInput , // Automatically sanitized }) ; Performance Integration ■ // Error reporting is performance-aware playerror . report (error , { performance : { loadTime : 1500 , memoryUsage : "high" , } , }) ; Compliance and Monitoring ■ GDPR Compliance ■ Data Minimization : Only collect necessary error context Right to Erasure : Error logs can be purged on user request Transparency : Users are informed about error reporting Security : Error data is encrypted in transit and at rest { "defaultMessages" : { "server" : "We're fixing it!" , "notFound" : "Item not found." , "auth" : "Please log in again." , "forbidden" : "You lack permission." , "timeout" : "Connection timed out." , "network" : "Check your internet connection." , "rateLimit" : "Too many requests." , "validation" : "Please check your input and try again." , "unknown" : "Something went wrong. Please try again." } , "piiFields" : ["password" , "email" , "token" , "apiKey" , "secret" , "authorization" , "cookie" , "sessionId"] } Migration Guide ■ From Console Errors ■ // Before console . error ("API call failed:" , error) ; alert ("Something went wrong") ; // After playerror . report (error , { component : "ApiService" , action : "fetchData" , }) ; playerror . showToast ("Unable to load data. Please try again." , "error") ; From Try-Catch Blocks ■ // Before try { const

result = await riskyOperation ( ) ; return result ; } catch (error) { console . error ("Operation failed:" , error) ; throw error ; } // After try { const result = await riskyOperation ( ) ; return result ; } catch (error) { playerror . report (error , { component : "RiskyComponent" , action : "riskyOperation" , }) ; throw error ; } From Global Error Handlers ■ // Before window . onerror = function (message , source , lineno , colno , error) { console . error ("Global error:" , error) ; return false ; } ; // After // Automatically handled by Play+ global error handler // No manual setup required Developer Checklist ■ Do I use playerror.report() or reportCritical() for all error handling? Do I provide component and action context with every error report? Do I show user-friendly error messages instead of technical details? Do I provide recovery options for every error scenario? Do I test error scenarios including network failures and invalid inputs? Do I use error boundaries for complex components? Do I monitor error rates and recovery rates? Do I avoid logging sensitive data in error context? Summary ■ The Play+ error handling system provides: Proactive Error Management : Catch and handle errors before they reach users User-Centric Feedback : Clear, actionable error messages Contextual Intelligence : Rich error context for debugging Security by Default : Automatic PII redaction and sanitization Recovery Paths : Every error has a clear way forward Monitoring Ready : Built-in metrics and alerting Compliance Ready : GDPR and security compliant Developer Friendly : Simple API that enforces best practices 90% of error handling is automated. Focus on the 10% that matters.